Design and Implementation of a Baseline port of the Jikes Research Virtual Machine to the IA-64 Architecture

by

Ravi kiran Gorrepati

B.Tech in Computer Science, 2002
Jawaharlal Nehru Technological University, Hyderabad

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2006

©2006, Ravi kiran Gorrepati

Dedication

To my parents

Acknowledgments

I express my deep appreciation for Prof. Darko Stefanović for strong emphasis on clear thinking, expert guidance and patience in working with me.

Design and Implementation of a Baseline port of the Jikes Research Virtual Machine to the IA-64 Architecture

by

Ravi kiran Gorrepati

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2006

Design and Implementation of a Baseline port of the Jikes Research Virtual Machine to the IA-64 Architecture

by

Ravi kiran Gorrepati

B.Tech in Computer Science, 2002
Jawaharlal Nehru Technological University, Hyderabad

M.S., Computer Science, University of New Mexico, 2006

Abstract

Of all 64-bit architectures, IA-64 is the most recent and offers exciting new features for compiler developers. IA-64 gains speed from efficient compilation, which is costly for Java, because the compilation of modules happens dynamically on the virtual machine. Lack of an opensource Java virtual machine for the IA-64 which allows for research on code generation techniques lead us to pursue this work. This work presents a baseline code generator for the IA-64 architecture. This code generator is a part of the Jikes Research Virtual Machine for Java written in Java. To our knowledge, this work presents the first opensource, just-in-time virtual machine for Java on IA-64 architecture

Contents

Glossary

GC	Garbage Collection
GCTk	Garbage Collection Toolkit
IMT	Interface Method Table
JTOC	Jikes RVM Table of Contents
JVM	Java Virtual Machine
RVM	Research Virtual Machine
TIB	Type Information Block
VM	Virtual Machine
Linux	The Linux operating system
MacOSX	The Mac operating system version X
PowerPC32	The PowerPC 32 bit architecture
PowerPC64	The PowerPC 64 bit architecture
IA-64	The IA-64 architecture

Chapter 1

Introduction

Of all 64-bit architectures, IA-64 is the most recent and offers exciting new features for compiler developers. Its features place it in the higher end of the RISC architecture spectrum. Since this architecture is very different from other extant architectures, it offers a lot of room for research. The presently available compilers do not perform as well as they should on this architecture.

The designers of the IA-64 architecture were motivated by the fact that Moore's law[1] is fast approaching its limit, and the known scale of improvement in speeds would not be possible in the future. Having multiple parallel processors is a solution, but the performance of single-threaded processes is unaffected by parallelism at the processor level. The IA-64 architecture provides parallelism at the level of functional units; it has multiple integer, floating point, and memory units. This enables the architecture to execute multiple integer, floating point and memory instructions simultaneously, *i.e.*, it provides inter-instruction parallelism. Inter-instruction parallelism exists in all of the out-of-order processors; these processors reorder the instructions in the instruction stream and issue more than one instruction per cycle. However, in the IA-64, this work of discovering the parallelism and re-ordering the instructions is pushed to the compiler. IA-64 is an in-order

[1]The empirical observation that transistor density on the chip doubles every 18 months.

processor, *i.e.*, it does not re-order the instructions in the instruction stream.

The IA-64 architecture, however, did not live up to the standards it was designed for. The adoption of IA-64 was slow because of its poor x86 hardware-emulation performance, a not so mature silicon implementation, the lower clock speed of the implementation, and relatively poor integer performance. The large digression of the IA-64 architecture from known architectures caused it to be poorly understood, and compiler performance suffered in consequence. The inter-instruction parallelism and the features to improve it put a large overhead on the compiler. The older architectures perform better, for out-of-order execution has been researched and implemented for a long time. Better algorithms that schedule the instructions at compile time should help the performance on IA-64. We believe that the architecture is good, and with time, better compilers for IA-64 will be available.

JikesRVM is an open-source research virtual machine for Java developed at IBM. The Jikes research virtual machine presently runs on x86/Linux, PowerPC32/ Linux, PowerPC32/MacOSX, PowerPC32/AIX, PowerPC64/Linux, PowerPC64/AIX, and PowerPC64/MacOSX. This work describes the port of JikesRVM to IA-64/Linux architecture. This project is a derivative of the working PowerPC64 implementation [**?**] developed in our laboratory. When this port of Jikes RVM to IA-64/Linux was started, there were very few commercial implementations of Java on IA-64. To our knowledge, there is still no open-source implementation of Java available on this architecture. This port, as we envisage, should enable research on improving the performance of Java on IA-64.

Chapter 2

Background

2.1 The Java Programming Language

The Java Programming Language [**?**] is a general-purpose, architecture-neutral, reflective, multi-threaded, strongly typed, memory-managed, class-based object-oriented language. It is similar to C++ in that it is also a class-based language. Its features, however, align it closer to Objective C than C++. It does not, for example, support operator overloading, multiple inheritance or automatic type coercion.[1] Java supports interfaces and dynamic method resolution. The specification [**?**] clearly distinguishes between compile-time and run-time errors. Compile time consists of generating the intermediate, architecture-neutral *bytecode*. Run-time consists of loading and linking of classes, machine code generation, and dynamic optimization. Java has exception handling, which provides a way to handle errors at run time. Reflection is supported by associating an object with a Class object. A Class object holds the metadata for an object. Architecture neutrality is provided by the *bytecode* and a strong specification of primitive types. The primitive types are strictly defined by the Java virtual machine specification [**?**] and thus behave similarly on all ar-

[1] Java 1.5 supports partial auto type coercion in the form of auto-boxing.

chitectures. Automatic reclamation of used space leads to fewer bugs in Java programs than programs written in other languages.

2.2 The Java Virtual Machine

The Java Virtual Machine [**?**] is a specification for an abstract stack based machine. The Java *bytecode* is the instruction set for this abstract machine. The specification [**?**] is independent of any architecture, operating system, or environment. The implementation method is left unspecified by the specification.

2.2.1 The Java bytecode

The Java Virtual machine specification [**?**] defines the set of instructions for an abstract stack machine, where instruction arguments are popped from an operand stack. After the execution of the instruction, the result of executing the instruction is pushed back onto the stack. Linkage errors or run-time errors that result from executing an instruction should be handled or thrown by the virtual machine accordingly, as defined by the virtual machine specification.

The bytecode instruction set broadly consists of loads, stores, method invocations, arithmetic operations, conditional control flow operations, and stack manipulation operations. Loads and stores can be to/from a local variable of a method, field in an object, class or an array. The method invocation instructions include virtual method invocation (*invokevirtual*), static method invocation (*invokestatic*), and interface method (*invokeinterface*) invocation. Arithmetic operations include addition, subtraction, multiplication, and division on all of the fixed and floating point types. The conditional control flow instructions consist of comparison operators on all primitive types. The result of the comparison instruction decides the flow of control. It should be noted that arithmetic and comparison

operations work on operands that are of the same type. The stack manipulation operations consist of pop, swap, and various duplicate instructions. Finally there is an instruction, *checkcast*, to do type casting and throw an error if the type cast operation is unsafe.

2.3 Jikes RVM

Jikes RVM is an open-source research virtual machine for Java developed at IBM T.J. Watson research center. It is intended to be easily extensible, modular, and object oriented, and is implemented in Java. Previous virtual machines written in Java ran on another VM as host machine [**?**]. Jikes does not need another VM to run on – it generates machine code for source, and the generated code for the VM can be run on hardware directly. Jikes thus is faster than all the previous VMs written in Java.

It must, however, be noted that Jikes needs another VM at build time. Thereafter, at run time Jikes RVM is already executing as machine code, and thus does not need a host VM. To put this in more detail,

- The Jikes RVM is written in Java. At build time, it is run on a host VM.
- A portion of Jikes RVM is a code generator. This code generator reads a class file and generates the corresponding machine code for the target machine.
- Feeding the code generator (which is written in Java) its own class files generates the machine code for the code generator. This is suitable for running on hardware.
- At build time, running on a host VM, the code generator generates the machine code for the entire VM.[2]

[2]Jikes RVM does not generate machine code for all the Java libraries. at build time; rather they are compiled at execution time as needed.

- A C code wrapper for Jikes RVM forms the interface to the operating system. It loads the generated machine code into memory and runs it.

Jikes RVM broadly consists of the following subsystems: core run time (class loaders, library support, scheduler, profiler, etc.); compiler (baseline, optimizing); memory managers, which include a variety of garbage collection mechanisms; and an adaptive optimization mechanism. All these subsystems are written in Java. The low-level functionality, which cannot be expressed cleanly in Java, is implemented using the services of the MagicCompiler, the details of which shall be described shortly.

2.3.1 Object Model

Objects fall into two categories, *viz.*, instances of classes and arrays. Class instances have a two word header, one word for the type and the other for synchronization and memory management.

One word of the header is called the *Type Information Block* (TIB) pointer. The TIB, itself an object, contains the class information, which applies to all objects of that class. It contains the virtual method table, a pointer to an object representing the type, and pointers to a few data structures to facilitate efficient interface invocation and dynamic type checking.

The second word is the *status* word, divided into three bit fields. The first bit field contains a pointer to a lock object, or is itself a direct representation of the lock. The second bit field contains the hash code for hashed objects, while the third bit field is used by the memory managers to store reference counts, forwarding pointers, etc.

An important difference between array objects and scalar objects (instances of classes) is that array objects grow up from the object reference, and scalar objects grow down.A null-pointer in Jikes RVM is represented as 0x0. The length field in an array is at an offset

of -8 from the actual reference. Accessing an element in a null array thus generates a hardware trap because all array accesses are checked for the bounds against the length of the array before accessing the actual element. The scalar objects grow down, and hence an access to a field of a null object accesses high memory and generates a hardware trap.

Object Internals

An object is an instance of a class. It consists of methods, fields, and other objects. An object is always accessed by its reference [**?**]. Objects internal to an object are references and hence fields. In JikesRVM, methods are arrays of instructions. Pointers to instance fields and methods are stored in the Type Information Block of the class.

The **invokevirtual** bytecode causes access to the TIB of the class to invoke a method. Access to methods of derived classes involve no complexity, for each method occupies the same slot in the defined class and the derived class. Overridden methods occupy the same slot, the difference being just that the slot element is a reference to the new method rather than the method defined in the base class.

The **invokeinterface** bytecode causes access to the *Interface Method Table* (IMT) [**?**], which is analogous to the TIB of the virtual methods. The IMT contains references to the interface methods, and just like the TIB, the derived methods occupy the same slot. The difference between the IMT and the TIB is that sometimes two different methods in the IMT may occupy the same slot, in which case a conflict resolution stub is inserted into the IMT.

2.3.2 JTOC

The *Java Table of Contents* (JTOC) contains references to all of the class fields and methods, string constants, numeric constants, literals and references to TIBs of all of the loaded

classes. [3] The JTOC is declared as an integer array, and may also contain references. A descriptor array co-indexed with it marks the references. In 64-bit architectures, the references occupy 2 array slots.

2.3.3 Class Loading

Jikes RVM implements the Java programming language's dynamic class loading. While a class is being loaded, it passes through six states in the following manner:

- **Vacant**

  The `VM_Class` object for this class has been created and registered and is in the process of being loaded.

- **Loaded**

  The class file has been read and parsed. The constant pool has been constructed. The declared methods and fields of the class have been loaded. The class's superclass and superinterfaces have been loaded.

- **Resolved**

  The superclass and superinterfaces of this class have been resolved. A list of the virtual methods and instance fields of this class, including the methods and fields inherited from its superclass has been constructed and the offsets for the instance fields have been calculated. Space has been allocated in the JTOC for all static fields of the class and for static method pointers and the appropriate offsets calculated. The TIB has been initialized and offsets for the virtual methods have been calculated.

- **Instantiated**

[3] A description of the class loading process is dealt with shortly.

The superclass has been instantiated. The slots in the JTOC are filled in with pointers to compiled code or lazy compilation stubs for the static methods. The slots in the TIB are filled in with pointers to compiled code or lazy compilation stubs for the virtual methods.

- **Initializing**

  The superclass has been initialized. The class initializer is being run.

- **Initialized**

  The superclass has been initialized. The class initializer has been run.

2.3.4 Magic

Magic is a mechanism to implement certain functionality that cannot be expressed in pure Java. Two types of Magic operators exist; the below seen example is one part of the first class. They are static methods of the **VM_Magic** class. The second type provides mechanisms to make certain portions of code *uninterruptible*.

To see the need for Magic, consider the following example which in the IA-64 port is used for debugging. The task is to get the value of the current Instruction pointer (IP) onto the stack. The Java language or its bytecodes do not have a notion of IP. However, each processor architecture has its special instruction, which allows it to read the value of IP. The Magic *getIP()* method, seen below, accomplishes this.

```
asm.emitADDS   (SP, -SIZE_ADDRESS, SP);
asm.emitMOVIPG (T0);
asm.emitST8    (T0, 0, SP);
```

The *getIP()* method stores the value of IP on the top of stack. This method can now be called from anywhere in the virtual machine, as a normal Java method. A println() method

in combination with getIP() method can thus be used to print the IPs at different points in the code, which is useful for tracking down bugs.

The methods in **VM_Magic** make operating system calls, perform unsafe casts, and implement accesses to raw memory. Since they cannot be expressed in Java, the bodies of methods in **VM_Magic** are undefined.**VM_MagicCompiler** contains Java instructions to generate assembly code for the methods defined in **VM_Magic**. When the compiler encounters the magic method, it inlines the appropriate code for the magic method into the caller method.

The second class of Magic operators, the *uninterruptible* methods, are compiled without the insertion of hidden thread switch points. Stack overflow checks and yield points will not be generated for *uninterruptible* methods.

2.4 The IA-64 Architecture

The IA-64 [**?**, **?**, **?**] architecture was designed from the ground up to address the limitations of existing processors. The IA-64 architecture allows the compiler to communicate with the processor, maximizing the use of functional units in the processor. The model is called Explicitly Parallel Instruction Computing (EPIC). The burden of finding the parallelism in the instructions lies with the compiler rather than the processor as with out-of-order processors.

2.4.1 Instruction Model

The IA-64 architecture supports instruction level parallelism (ILP). ILP is the ability to execute more than one instruction at the same time. To facilitate such a mechanism, IA-64 groups three instructions into a bundle. In general, the hardware may execute all three instructions in a bundle concurrently. The compiler can explicitly prevent concurrent ex-

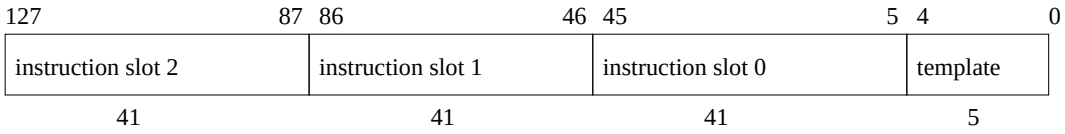

Figure 2.1: Bundle Format

ecution of instructions with a stop bit. The stop bit indicates the location in the bundle where inter-instruction concurrency breaks.

Concurrent execution mandates that instructions follow rules with regard to available resources. All three instructions can read from the same register or memory location, but none of them should write while other instructions are reading from the same source. For more details on instruction sequencing, see Section 3.4 of Itanium Architecture Software Developers Manual [**?**].

A bundle is 128 bits long, 5 bits of which are allocated for the bundle template. The instructions themselves are 41 bits long. The bundle template specifies the functional units the instructions should run on and the location of the stop bit. For more details on instruction encoding, see section 3.3 of Itanium Architecture Software Developers Manual [**?**].

2.4.2 Execution Model

In true RISC tradition, the IA-64 supports large register sets. This allows the compiler to exploit ILP easily. The IA-64 architecture offers 128 general-purpose registers (GPRs), 128 floating-point registers (FPRs), 64 predicate registers (PRs), 8 branch registers (BRs), and 128 application registers (ARs). The GPRs, BRs, and ARs are all 64-bit wide. The FPRs are 82-bit wide and offer extended precision. The predicate registers are 1-bit wide.

General Purpose Registers

The GPRs, numbered from 0 to 127, are the principal resource for integer arithmetic. GPR 0 is special and always has the value 0. The GPRs are divided into two subsets. The first 32 GPRs are *static* while the rest are *stacked* registers. Stacked registers, also known as the register stack, prevent spilling registers to memory at function boundaries, by renaming the registers. The registers used by a function in the register stack are divided into input, local, and output registers. The local and input registers contain the local state of a function. When a function A calls function B, the parameters to function B are placed in the output registers of A. During the function call, the output registers of A are mapped to input registers of B. When function B returns, the local state of A is restored by remapping the register set to the original configuration. When the register stack is full, it is written to backing store in memory. For more details on register stack operation, see section 4.1 of Itanium Architecture Software Developers Manual [**?**].

Floating Point Registers

The FPRs are 82 bits wide and are numbered from 0 to 127. They support extended precision. FPRs 0 and 1 are special and always have values +0.0 and +1.0 respectively. FPRs 32 to 127 are rotating floating point registers and can be programmatically used to accelerate loops.

Predicate Registers

The Predicate registers are 1-bit wide and are numbered from 0 to 63. Predicate register 0 always has the value 1. Predicate registers, being 1-bit wide, can hold only boolean values. Instructions in the IA-64 have provision for a predicate register. Execution of an instruction can be predicated by the register, *i.e.*, the instruction will be executed only if

the predicate register holds the value 1. Registers PR16-63 are rotating predicate registers and can be used to accelerate loops.

Application Registers

The Application registers provide control over managing the register stack, backing store, and other counter registers. They also provide precision control over floating point operations, and a register, *ar.ccv* for use in XCHG (exchange) instructions. The XCHG instructions are crucial for atomic operations.

2.4.3 Architectural Conventions

GPR conventions

r0 Register r0 always has integral value 0. Writing to this register generates an illegal instruction fault.

r1 Register r1 is the global pointer, and points to the current global data segment. The data segment holds local variables of a procedure.

r2, r3 These are the only registers that can be used in the 22-bit add instruction

r4-r7 These registers are preserved registers and should be saved by the callee if used.

r8-r11 These registers hold the return values from a procedure.

r12 This register is the *stack pointer* in the IA-64. It points to the address of stack's topmost valid word. The *stack pointer* must always be aligned to 16-bytes.

r13 Also known as the thread pointer, this should not be modified by application programs.

r32-r39 This subset of registers is where the callee procedure finds its input parameters.

For a more detailed treatment of GPR conventions, see section 5.2 of IA-64 conventions guide [**?**].

Chapter 3

Design

In this chapter we discuss the design decisions taken for the present implementation of the IA-64/Linux port of Jikes RVM. First of all, the primary goal is to a have a simple working port, which is similar to the PowerPC 64-bit implementation. Design decisions about object layouts, mapping IA-64 register conventions to those in Jikes RVM, instruction bundling or the lack of it, stack frame layout, system calls, and signal handling follow.

3.1 Object Layout

The Java Programming language [**?**] broadly operates on two kinds of types, primitives and objects. Primitives consist of booleans, Unicode characters, integers of all sizes, and floating point numbers of IEEE754 single and double precision. The objects consist of class instances and arrays.

An object consists of a header and a body. The body of the object contains all its members (fields and references to methods). The object header consists of a reference to its type information block and a status field. The TIB of the object describes the class, superclass, and interfaces it implements, and contains pointers to its virtual methods. The

contents of the status field tell whether or not the object is locked. The status field also holds the forwarding pointer during garbage collection. Note that the TIB and status field (during GC) need to hold pointers to objects. Both these fields thus need to be 64 bits long, making the object header 2 words[1] long. Furthermore, all objects are aligned to word boundaries. This ensures that the status field is aligned to a word boundary, for loads and stores in IA-64 are atomic only on word boundaries. Atomic access to the status field is necessary for object locking. The object layout for IA-64 thus remains exactly the same as for PowerPC64, except for arrays.

Methods in Jikes RVM are arrays of instructions. By architectural conventions [**?**], instructions in the IA-64 should always be aligned to 2-word boundaries. The default header size for arrays in PowerPC64 is 12 bytes. The actual array contents start after the header. Aligning the start of array object to 2 words does not guarantee that array contents are aligned to a 2-word boundary. To handle that, we added a 4 byte null word to the array header making the header size 2 words long.

3.2 Bundling

As discussed earlier, bundles in the IA-64 contain three instructions. Our implementation has only one instruction per bundle. The other two instructions in the bundle are no-operations (nops). Having nops in the instruction stream results in rather large binaries and affects cache performance. However, having more than one instruction per bundle requires dependency analysis, doing which blurs the distinction between a simple baseline compiler and an optimizing compiler. From the implementation point of view, especially with Jikes RVM's building model, it is difficult to get it right when porting to a new architecture, all the while re-arranging the instructions depending on the parallelism between them. The present IA-64 port of Jikes RVM has poor performance, and most of it can be

[1] A word is the length an of address in bits, on the target architecture. The IA-64 is a 64-bit architecture machine, and from here on 64 bits are referred to as a word.

attributed to not filling the bundle. Furthermore, the IA-64 architecture is in-order, thus resulting in execution of no more than one instruction per clock.

The bundle templates *MFI_*, *MFB_*, and *MLX_* are the only ones considered for bundling, and in that order. In the bundle templates, *M* indicates a memory instruction (loads and stores), *F* indicates a floating point instruction, *I* a fixed point instruction, *B* a branch instruction, and *LX* a long-extended instruction (used for loading 64-bit values and branching to 64-bit addresses.). The _ at the end of the templates indicates the stop bit, which prevents consecutive bundles from being issued in the same clock. For information on more bundle templates see Section 4.0 of Itanium Software Developer's Manual [**?**].

3.3 Program Stack Frame layout

Program stack frame layout in Jikes RVM is dictated by the virtual machine specification [**?**]. A thread in Java has an associated JVM stack. Methods in the thread allocate frames for storing operand stack and local variables.

The local variables of the method are referenced via a fixed offset from the frame pointer (FP). The offset of the variable is known at compile time. The IA-64 architecture mandates (for efficiency reasons) that all fields are aligned on their natural boundaries, *i.e.*, a 32-bit entity should be aligned to 32-bits in memory. In the PowerPC64 port of Jikes RVM, all fields, regardless of their size, are aligned to 64-bits. Since all Java primitive types measure less than or equal to a word, not more, this design works well with the IA-64 port and remains unchanged. Aligning word length entities to a word also ensures that atomic access of those fields is possible. However, it should be obvious that allocating word-sized slots for smaller entities wastes heap space and could cause frequent garbage collection. Aligning all slots to their natural boundaries, while not hurting cache performance, is left as future work.

As mentioned earlier, each method frame carries its operand stack. The location of top

of the operand stack is contained in the stack pointer (SP) register. At each operation, the operands are read and the results written back to the stack while updating the stack pointer. Later versions of Jikes RVM do not have a stack pointer because the position of the stack pointer is known at compile time for every bytecode in a method. However, whether the stack pointer register can be removed from the IA-64 Jikes RVM when updated to the later versions remains to be seen, for the IA-64 architecture does not support displacement addressing mode. It may be possible that those effects can be minimized when bundling more than one instruction is achieved.

In the PowerPC64 port, all slots on the operand stack are of word length. This is to make sure that untyped operands such as, dup and dup2, work without paying heed to how long the actual operand is. This does not incur much space, for operand stacks are typically small. Since all operands are allocated word length anyway, we decided to have sign-extended 64-bit values for all the fixed point primitive types in the IA-64 port. A value written to the operand stack is always sign-extended before doing so. This is efficient because the load instructions in the IA-64 do not do automatic sign extension. A sign-extension instruction should follow the load operation whenever appropriate. Correctness of working with 64-bit sign-extended values on the stack is ensured, because addition, subtraction, and multiplication operations do not affect the lower-order bits even in the case of overflow. Only during division, boolean shift, upward conversion (e.g., int to long) and array access operations are the appropriate lower-bits read from the stack and sign-extended (if necessary), for reading the entire 64-bit value may not be correct.

3.4 Register Mapping

Register usage should respect the conventions of the target architecture so that the generated code works correctly and importantly works with system calls and signals. Furthermore, registers are used in Jikes RVM to minimize saving/restoring them across method

calls and system calls. The baseline compiler so closely follows the stack[2] that it is not even necessary to save all the registers during method calls!

3.4.1 Preserved General Purpose Registers

By IA-64 run-time conventions [**?**] general purpose registers r4-r7 are preserved. Preserved registers are those which are not saved by the caller, but the callee restores their value, should it use them. Preserved registers are also called non-volatile registers.

Jikes RVM internally has the JTOC, Java table of contents structure, which contains references to all class (static) members, string constants, numeric constants, and references to all TIBs in the virtual machine. Since this is a heavily used structure, a register JTOC points to the start of this array. Since the JTOC structure is shared across all the classes and methods, the preserved register r7 is used for it. The SP, stack pointer in Jikes RVM, keeps track of the top of the stack in each method frame. Preserved register r5 is used for the SP. The PROCESSOR_REGISTER in Jikes RVM tells us what processor the present thread is running on. The preserved register r4 is used for it. The THREAD_ID_POINTER contains the identification for the present thread. This register is used during object locking. The preserved register r6 is used for THREAD_ID_POINTER.

The frame pointer(FP) in Jikes RVM points to the first word of the frame, which holds the previous frame's frame pointer. The frame pointer is needed for unwinding the stack frame during epilogues. The preserved register r12 is the stack pointer by IA-64 run-time conventions [**?**]. Though named differently, this register by IA-64 conventions performs the same action as the frame pointer in Jikes RVM. It is not only natural to map the frame pointer to r12, but necessary, for it holds the pointer to the last word of the present frame, beyond which the frames for system calls and signals are built on(And thus respecting the C runtime conventions is necessary.). By convention, it is also necessary to

[2]The state of the running method is stored entirely on the stack

make sure that this register always is aligned to 2-words. All the above registers in Jikes RVM are mapped to preserved registers in IA-64 for these need not be saved/restored during system calls. The values in these registers live long and are only changed during thread switching (THREAD_ID_REGISTER, frame pointer), processor switching (PROCESSOR_REGISTER) and method prologues and epilogues (frame pointer).

3.4.2 Other General Purpose Registers

The registers r2, r3, and r14-r31 are scratch registers by IA-64 run-time conventions. Scratch registers are those that are saved by the caller, and restored when the callee returns. Scratch registers are also called volatile registers. The registers r2, r3, r14...r16 are used as volatile registers in Jikes RVM and are mapped to Jikes RVM's baseline compiler register names T0...T5 respectively.

In addition, we defined a new class of registers called Miscellaneous which are used in prologues and epilogues. Our port uses these registers to perform the same functions performed by registers 0 and S0 in the PowerPC64 port . These registers are only used in prologues and epilogues, and are not saved across method calls.

3.4.3 Floating point registers

The floating point registers f16 to f21 map to registers F0...F5 respectively in Jikes RVM. Register f22 is used as the Miscellaneous floating point register. This register is used for temporary storage during prologues and epilogues.

3.4.4 Predicate registers

Predicate registers are registers that hold boolean values. As said earlier, execution of an instruction in IA-64 can be predicated. These registers hold the result of compare instructions. Registers p6-p15 by IA-64 run-time conventions are volatile, and the first five of those map to registers P1, ...P5 respectively. We also have two Miscellaneous predicate registers, which are meant to be used in prologues and epilogues. However, it should be noted that all the predicate registers are saved/restored at once using broad side access.

3.4.5 Branch registers

Branch register b0 is scratch (volatile) by IA64 conventions. We used this register as the link register (LR) is used in PowerPC64. After a call instruction, this register, by convention, holds the return address. It is mapped to register BS (branch scratch) in Jikes RVM.

3.4.6 Application registers

Application registers in the IA-64 hold error codes, counters, and other special values. The Application Registers used in Jikes RVM are ar.ccv(compare and exchange value register) and ar.pfs (previous function state). The first register is used with the CMPXCHG (compare and exchange) instruction. This instruction is used in monitors and spinlocks. The previous function state register is not necessary to us, as we did not use the register stack. The *alloc* instruction, which handles the register stack, however, forces us to save this register and thus we save it.

3.5 Register Stack

Our implementation uses at most five scratch GPRs per method. Using a register stack nullifies the time spent on passing parameters. Rather than writing parameters to memory, the register stack allows the compiler to pass parameters in the registers. For simplicity, however, our present port of JikesRVM does not use the register stack. Also, note that the other register sets (floating point for example) do not have stacked registers and we end up writing them to memory at method boundaries. However, the Jikes RVM Quick compiler could make very good use of the large register set of IA-64.

3.6 System calls

Not using the register stack is unacceptable during system calls because the IA-64 C run-time conventions mandate that the parameters are passed through the register stack only. During the system call we allocate the appropriate number of registers on the register stack and load the parameters into those registers. Registers are allocated on the register stack with the *alloc* instruction. After the system call, by run-time conventions, registers r8 . . . r11 hold the return value. On returning from the system call, the register stack of the previous function is restored and the return value is placed on the top of the stack.

3.7 Signal Handling

The signal handler for Jikes RVM is written in C, and takes any necessary action in response to the event that generated the signals. Some of the signals are converted into Java exceptions, which are sent to the VM. When an event cannot be handled, the stack trace is printed with a proper description of the error that caused the crash.

As we have seen earlier, in Jikes RVM, both the volatile and non-volatile IA-64 registers are used. By C run-time conventions, the volatile registers on IA-64/Linux are saved in the *pt_regs* structure by the caller. The non-volatile registers are not saved during a method call. Should the callee use the non-volatile registers, the registers are first saved in the *unwind descriptor list* data structure. According to the conventions, every method frame in IA-64 should have an *unwind descriptor list* data structure. On method return, the *unwind descriptor list* is consulted to restore the non-volatile registers.

When a signal-generating event occurs the control passes to the kernel. For a detailed explanation of what happens when a signal occurs on IA-64 refer to the IA-64 Linux kernel book [**?**]. The values of the non-volatile register might have changed because the kernel uses them. To see how we get access to non-volatile registers, lets first see how the signal handler works on IA-64/Linux. The POSIX conventions dictate that the signature of a signal handler is,

```
void sighandler(signum)
```

Linux does provide such a facility, but also provides much more. It provides the interface

```
void sighandler(signum, siginfo, sigcontext)
```

The third argument, sigcontext, is nothing but a copy of the registers saved before the kernel is entered *i.e.*, the *pt_regs* structure. Thus the *sigcontext* contains all of the volatile registers. The non-volatile registers are accessible only from the *unwind descriptor list*. Access to this data structure is provided by the unwind library. We unwind through the signal stack until we reach the *signal frame*. The *signal frame* is the first frame created in response to the signal generating event. Unwinding deeper throws us into user code, which is not safe, because Jikes RVM internally does not conform to all the run time conventions. For example, the frames in Jikes RVM do not have an associated *unwind descriptor list*. Unwinding through the kernel frames until the signal frame gives us the values of non-volatile registers active at the time of signal generating event.

Chapter 4

Implementation

In this chapter, we discuss the implementation of the Jikes RVM port to the IA-64 architecture. Our implementation deliberately does not use features of IA-64 such as predication, instruction bundling, register stack, control speculation, and data speculation. The porting effort requires writing all the architecture-specific parts of the virtual machine before one begins to test them. JikesRVM during the build process has to be compiled by its compiler; thus it is not possible to test the compiler on small, well-understood pieces of code early on. It is thus not easy to implement poorly understood features of the target architecture in the initial port of Jikes RVM. Additional features of the IA-64 can, however, now be added and tested incrementally. Getting Jikes RVM to boot was rather demanding, because of the large amounts of new untested code in a much larger context.

This work is derived from the PowerPC64 port of Jikes RVM [**?**]. That port did not have the JNI implementation, adaptive compilation, or the 64-bit ported ”Watson” (IBM) collectors. We exclude these features in this port.

JikesRVM, at the source code level, consists of architecture-dependent code, which is the low-level compiler implementation, and an architecture-independent portion, which forms the rest of VM including the garbage collectors. The architecture dependent and independent portions are separated; the code generators for different architectures are

entirely different and are organized into files in different directories. Writing the code generator forms a large portion of this work. There are, however, portions of code in the architecture-independent sections, such as thread switching and exception handling, which need different treatment depending on the architecture. This chapter describes the code generator, which includes the assembler and the compiler, the wrapper C code, boot image writer, the system calls, and signal handling.

4.1 The Code Generator

Three compilers, ranked by optimization, form the code-generator section of the VM. The baseline compiler insists on faster compilation, rather than generating optimal code. Its performance is close to that of an interpreter. The optimizing compiler generates optimized code at the cost of increased compilation time. Somewhere on the performance scale between these two compilers lies the Quick compiler. The Quick compiler makes use of the large register sets provided by the modern RISC architectures. Its compile performance is similar to that of the baseline compiler, but the run time performance of the generated code is much better. Finally, Jikes RVM has an adaptive compiler, which based on the profile data, makes use of the compilers described above. As said earlier, the present implementation has only the baseline compiler.

4.1.1 The Assembler

The assembler *VM_Assembler* generates the appropriate bit patterns for assembly instructions. For example, the *st8* instruction which stores an 8-byte entity (in a source register) to a memory location pointed by the target register, is generated by the assembler like this:

```
static final INSTRUCTION ST8template = 5L<<37
```

```
                          | 51L<<30 | 0<<28 | PR0 ;

  static final INSTRUCTION ST8 (int r3, int imm, int r2){
  return ST8template | ((long)imm>>31 & 0x1)<<36
        |(imm>>7 & 0x1)<<27 |r3<<20|r2<<13|(imm&0x7F)<<6;
  }

  final void emitST8 (int r2, int imm, int r3){
  //st8.none.none [r3]=r2,imm
  if (VM.VerifyAssertions) VM.assert(fits(imm, 9));
  INSTRUCTION mi = ST8template | ((long)imm>>31 & 0x1)<<36
                 | (imm>>7 & 0x1)<<27|r3<<20 | r2<<13
                     | (imm&0x7F)<<6;
  if (VM.TraceAssembler)
     asm((long) mIP, mi, "st8", "r", r3,
               "r", r2,  signedHex(imm), true);
  mIP = mc.addInstruction(mi, 'M');
 }
```

The emitST8 method generates the bit pattern for the ST8 instruction and adds it to the instruction stream with the help of the *mc.addInstruction* method. Since instructions in the IA-64 exist in bundles, it is necessary first to bundle the instruction.

The Bundler

As said earlier, only one instruction is placed in a bundle, which can otherwise take three. The second argument of the *addInstruction* method, *M*, indicates that the instruction should be run on the memory unit. This information is passed on to the bundler, which

bundles this instruction with appropriate nops, because the IA-64 [**?**] allows only a certain number of valid bundle templates.[1] Bundling is completed by writing the bundle template to the bundle. The bundle types *MFI_*, *MFB_* and *MLX_* are the only ones considered for bundling, and in that order. The bundler consists of a *case* statement, which builds the bundles according to the instruction type. A bundle is 128-bits long; since there is no integral type that is longer than 64-bits in Java [**?**], two long entities *hi* and *lo* constitute the higher 64-bits and the lower 64-bits of the bundle respectively. The bundler allows access to these *hi* and *lo* parts of the bundle through *getHi* and *getLo* methods. The *addInstruction* method gets the *hi* and *lo* entities of the bundle and adds them to the instruction stream. Note that the instruction stream is an array of longs in the IA-64, which in the PowerPC was an array of integers.

4.1.2 The Compiler

The compiler *VM_Compiler* is a big *case* statement that generates the appropriate IA-64 instructions for the Java bytecodes. Most of the bytecode instructions are simple and have simple IA-64 instruction expansions. There are, however, some obvious differences between the IA-64 and PowerPC instruction sets, which are reflected in the compiler.

The IA-64 does not support displacement addressing mode, *i.e.,* it does not allow memory operations (loads and stores) from an address in a register with an offset added. With the stack pointer, which points to the top word of the frame, this resulted in incrementing the stack pointer first with an *ADD* instruction and then storing the value at the updated address. This in the PowerPC could be done with a single *store* instruction, which could add the offset and store at the same time. The IA-64 architecture allows updating the address register after storing/loading the value, however. The PowerPC allows this

[1]The bundle template describes the types of the instructions that can constitute a bundle. The type of an instruction indicates the functional unit the instruction should run on.

only before the memory operation. To make the code generator efficient, these changes required us to re-order the instructions so that the generated code does not spend time in explicitly adding the offsets. Sometimes it was necessary for us to load the offsets using a *MOVL* instruction which stores a 64-bit value in a register. This was necessary because the IA-64, unlike the PowerPC, does not support adding immediate constants to the upper 16-bits of a register. The other reason was that the two flavors of *ADD* instruction, the *ADDS* and the *ADDL* instructions, support immediates that are up to 14 and 22 bits long only. The Java virtual machine specification [**?**] in many cases requires loading from 32-bit offsets. The problem with a *MOVL* instruction is that it measures two instructions in a bundle. This does not result in a performance hit in the present baseline compiler, but will when the bundling feature is supported.

As we have stated earlier, a bundle exists as two long values in the instruction stream. The IP (instruction pointer) relative form of branch instruction in the IA-64 requires the relative offset that is quantified in bundles rather than instructions. Note that this has a side effect of the branch target always being the first instruction in the bundle; it is not possible to jump into an instruction in the bundle other than the first instruction. This is not a problem for us, for our bundles contain only one instruction. However, representing a bundle as two long values resulted in our handling the forward branches differently. In the normal case, the difference between the branch instruction and the branch target gives the offset, which when the target instruction is being generated, is encoded into the branch operation. Note that this difference is not the correct offset; the difference is twice the actual offset, for two longs constitute a bundle.

The synchronization primitives are handled with reservation based stores in the PowerPC64. The IA-64 architecture does not support reservations, however. The lock value, usually in the object header, is first stored in the application register *ar_ccv*. The object header is then checked for a lock or the lack of it, and an appropriate new value is computed in a general purpose register. Now the object header value is checked for a change in value by comparing it to *ar_ccv* register, and if not changed, the new value is written to

the header. These compare and store operations happen in a single atomic operation made possible by the *cmpxchg* instruction. After the *cmpxchg* instruction the value that was to be written to the header and the value in the header are compared for equality, and if they are not equal, an appropriate action is taken.

The other important difference was that the IA-64 does not have separate instructions for floating and integral division. Even integral multiplication has to be handled by the floating point unit by loading the integral values into floating point registers.

4.1.3 The Magic Compiler

The magic compiler *VM_MagicCompiler* provides direct access to memory for the garbage collectors, provides the interface for making system calls, and does unsafe Java type conversions in a controlled way.The entire magic compiler has to be ported because it is completely architecture-specific. However, most of the porting effort here is straightforward except for system calls, which we discuss here.

System calls

The IA-64 architecture, by C run time conventions, requires that method arguments always be placed on the register stack. In Java code, we haven't used the register stack, however. The following code shows how system calls are made in IA-64.

```
if (methodName == VM_MagicNames.sysCall_AI_rI)
{
asm.emitB_CALL (BS, 1);
asm.emitALLOC  (32, 0, 1, 2, 0);
asm.emitLD8    (34, SIZE_INTEGER, SP);   // load value
asm.emitLD8    (33, SIZE_ADDRESS, SP);  // load address
```

```
generateSysCall(asm);
generateRemoveRegisterStack(asm);
generateSysCallRet_I(asm);
return;
}
```

The Magic method sysCall_AI_rI takes as input parameters an address and an integer, and returns an integer. The magic methods are inlined into the generated code by the magic compiler *VM_MagicCompiler*. The *alloc* instruction creates the space for the parameters on the register stack. The last four parameters of the *alloc* instruction indicate the number of input, local, output, and rotating registers needed for the caller method. Here we have asked for 0 input registers, 1 local register, 2 output registers, and 0 rotating registers. The *alloc* instruction stores the previous function state contained in application register *ar.pfs* in local register r32. The load *ld8* instructions load the parameters into output registers r33 and r34. Note that after the call, the caller's output registers are mapped to the callee's input registers. For more information about register renaming, consult Chapter 6 of Itanium Software Developers Manual [**?**]. The B_CALL instruction makes a call to the next instruction. This was necessary for setting up the register stack. The method *generateSystemCall* generates the code for making the system call. On return, *generateRemoveRegisterStack* removes the register stack just created and restores the previous function state. The return value, by convention, is contained in the register r8. The Magic method, *generateSysCallRet_I* moves the return value of the system call from register r8 to the top of the stack. The PowerPC64, by runtime convention, requires that method parameters are passed through a new frame allocated on the stack for the system call. It is not the case in IA-64 that we require a new frame, so we removed that during the port.

4.1.4 Traps and Signal Handling

In Java, access via a null pointer would result in a null pointer exception being thrown. Since the object header is at a negative offset, accessing a null object results in a segmentation fault at a very high address. The signal handler checks for high address access when a segmentation fault occurs; if it is a high address access the signal handler calls the Java code that throws a null pointer exception. Array indexes, on the other hand, are first checked by the compiler and, in case of illegal access, a user-defined hardware trap is generated. The hardware trap is then caught by the signal handler and converted to an improper array bounds access exception. In IA-64, traps with trap codes in range 0x80000 - 0xfffff are user-definable.User defined traps in Jikes RVM are used to signal improper array accesses, division by zero, and stack overflow.

As discussed earlier in the design section, the signal handler cannot access the values of preserved registers active at the time of the signal generating event. Note that the signal handler's frame is not the first frame that is created in response to the signal generating event, and thus, the preserved register values that we see there are not the values that were initially when the trap occurred. To handle that, we must unwind the stack frames until the signal frame.[2] The IA-64 unwind library [**?**] allows unwinding the stack frame seamlessly, provided all the stack frames conform to strict IA-64 conventions. The *unw_step* function of unwind library allows one to step back through the frames, one step at a time. The function *unw_is_signal_frame* tells whether the present frame is the signal frame. Once the signal frame frame is reached, it is straightforward to get the values of the registers with *unw_get_reg* function. The unwind library provides many more functions to walk through the frames; it even allows one to set the values of the preserved registers. For more information on unwinding, refer to *libunwind* man pages [**?**].

[2]The signal frame is the first frame that is created on the stack in response to the signal generating event.

4.1.5 Boot Image Writer

Porting the boot image writer was mostly straightforward; It only involved writing the values to the boot image in little endian rather than big endian as in PPC, because the IA-64 natively is a little endian architecture. One other change was that all arrays are aligned to 2-words when written to the boot image file.

Additionally, the IA-64 conventions result in writing the C function pointers to the boot image differently. In Jikes RVM, the first Java object, the boot record, is the communication area between the host operating system and the virtual machine. It consists of read-only values that are useful during startup. It also consists of operating system call interface methods. In most other architectures, getting the C function address involves

```
function_address = (ADDRESS) function
```

On the IA-64 the function pointer obtained thus does not point to the first byte of the function code. Rather, it points to a 2-word structure that describes the function. The first word in the structure points to the first byte in the code, while the second word points to the address of that function's globally addressable data segment. Getting the actual function address thus involves an indirection:

```
actual_function_address = *((long *) function)
```

Chapter 5

Validation and Benchmarking

Most of the work discussed until now involved porting the code generator. The code generator generates the code for the target architecture, in this case, the IA-64 architecture. It is necessary to test the code generator on a range of programs so that one may know with reasonable confidence that this code generator works on all the Java programs.[1]

We have initially tested Jikes RVM with the standard bytecode tests that are included with the Jikes RVM sources. Most of the debugging time was spent in booting Jikes RVM. Initially we included the bytecode tests in the boot image, which run every time Jikes RVM boots. We have also written our own set of tests, which initially were included in the boot image.

[1] Jikes RVM version 2.0.3 on PPC64 does not run all the Java programs, because of the lack of JNI support and class libraries. Here, in testing, we restrict ourselves to the same subset of Java programs that the PowerPC64 port was able to run.

5.1 Testing

5.1.1 Bytecode Tests

The bytecode tests test all the bytecode instructions in the Java VM specification [**?**]. This set of tests is not exhaustive in that it does not test a bytecode instruction with all possible inputs. Real world programs may not run if the bytecode tests are not passed. As said earlier, we included the bytecode tests into the bootimage. This was helpful in not only identifying bugs that obstruct booting the virtual machine, but also in identifying bugs that were not readily visible because some of the bytecodes have not been encountered, or may not have been used the way they were in the tests until that point in the boot process.

5.1.2 Our Tests

The set of tests we have written test for appropriate control flow for all the control flow structures available in Java [**?**], arithmetic operations for integral and floating point operands, and type conversions on primitive types. Control flow tests tested for proper execution of for loops and while loops with different increments and switch-case structures. This set of tests are basic, but they do show one that control flow structures are not to blame in the case of bugs.

The second set of tests, the arithmetic operation tests, were exhaustive and stressed the code generator. As stated earlier in the design section, we used an entire stack slot for integers and other primitive types which measure less than the slot. It is easy to have bugs in cases that did not handle overflow correctly. Thus, the arithmetic tests stressed the overflow condition with a range of tests like addition, subtraction, multiplication. division, comparison, left and right shifts, bitwise operations and negation operation. This set of tests was conducted for short, int, and long primitive types. Also, the mixed arithmetic operations, which involve auto-conversion of primitive types were tested. Finally,

the arithmetic operations tested the floating point values with addition, subtraction, multiplication, division, and trigonometric operations available in the Java Math library. This set of tests are included with the Jikes RVM source, so that one may test the IA-64 code generator when changes are made.

5.1.3 Other Standard Tests

This set of tests included the SPECJVM98 benchmarks and the set of tests that are included with the JikesRVM sources that test reflection, serialization, and threading. The SPECJVM98 benchmark has tests that check whether the VM works according to the specification [**?**]. These included loading and instantiation of new classes, interfaces and objects, all varieties of method invocations supported by Java [**?**], and exception handling. Since these tests are readily available here we did not write tests for these cases.

5.2 Benchmarking

Benchmarks included the *jess*, *jack*, and *javac* benchmarks that are included in the specJVM98 benchmark suite. The heap size is set to 100 MB for all the benchmarks. The IA-64 machine is an Itanium-2 dual core (of which only one processor was used), running at 900MHz with 8 GB memory installed. The operating system is RedHat Linux 7.2, the version of the Linux kernel being 2.4.18. For the sake of performance validation, we compare the results against those obtained on PowerPC64. The PowerPC64 machine is a PPC970 running at 1600MHz with 1GB memory installed. The operating system is YellowDog Linux 1.12, the version of the Linux kernel being 2.6.1. All the benchmarks were run with assertion checks disabled in JikesRVM. Table 5.1 shows the benchmark results in seconds. These results show that performance of Jikes RVM on IA-64 is competitive with

Table 5.1: JikesRVM on IA-64 - Comparison with JikesRVM on PPC64

Benchmark	SemiSpace/IA-64	SemiSpace/PPC64	Gen/IA-64	Gen/PPC64
jess	113.9	81.7	110.2	75.5
jack	106.5	70.7	103.5	69.1
javac	185.4	124.7	173.2	120.5

that on PowerPC64. The performance difference is attributed to difference in the clock speed of respective machines.

Chapter 6

Conclusions

Porting JikesRVM is a demanding task because it requires writing all the code before one begins to test it. Debugging it is not easy as there are few tools that could be used on this architecture. However, the good result is that its build model forces one to write correct code in the first place.

Porting JikesRVM to the IA-64 architecture provides a number of exciting research opportunities in code-generation on this architecture. To our knowledge, this work introduces the first open source Java VM on IA-64.

Chapter 7

Future Work

It is now necessary to port JikesRVM to the present CVS head. One of the very important changes would be removal of the stack pointer *SP* from the code generator. It is not necessary to keep track of the stack pointer in a register because the position of the stack pointer is known at compile time for all the bytecodes. This was a good idea in PowerPC, for that architecture supports displacement addressing mode. Since the IA-64 does not support displacement addressing mode, it may be a good idea to use IA-64 register stack for a method's operand stack. This would remove unnecessary loads and stores at each bytecode instruction, all the while not using the stack pointer register. Note that such an action can give a simple Quick compiler for JikesRVM.

Appendix A

Building Jikes RVM on IA-64/Linux

Building Jikes RVM on IA-64/Linux is similar to building it on other architectures. This document should be used in conjunction with the userguide[1] provided with Jikes RVM sources. Here we describe how to build Jikes RVM 2.0.3 using the crossbuild technique. For the crossbuild, the build process consists of two steps viz., building a boot image on the host machine, and building an executable Jikes RVM on the target machine . The host machine can be IA-32 or PowerPC-32/64 running Linux. The target machine is Itanium2 running Linux.[2] Though building Jikes RVM entirely on IA-64 is theoretically possible, it is not done so here, for when we started working on Jikes RVM there were no Java virtual machines available for IA-64 architecture. Access to the following packages will enable one to build Jikes RVM.

- The RVM source distribution. Jikes RVM 2.0.3[3] for IA-64 from Oal[4] CVS.

[1]The userguide is available in the directory ravi-ia64/rvm/rvm/doc
[2]The test machine runs Red Hat Linux 7.2
[3]Available as ravi-ia64
[4]Object Architectures Laboratory at UNM

- The RVM libraries. The Java libraries are included with the RVM source distribution[5].

- Other Prerequisites. The following tools are required to build Jikes RVM. Note that if one is not cross-building Jikes RVM, the host and target machines are one and the same.

  – make. The GNU make tool for both the host machine and target machine.

  – ksh. The korn shell, ksh-93 from AT&T. This can be downloaded from `http://www.research.att.com/~gsf/download/`. This is required for both the host machine and the target machine.

  – Java VM. The IBM Developer kit available at `http://www-128.ibm.com/developerworks/java/jdk/linux/download.html`. It is necessary to get a Java VM that supports only Java 1.3 since Jikes RVM 2.0.3 runs only that version of Java. This is only required for the host machine.

  – Jikes compiler. The Jikes bytecode compiler version 1.13. It can be downloaded from `http://jikes.sourceforge.net/`. This is only required for the host machine.

  – Unwind Library. Unwind library v0.97 for the target IA-64/Linux machine. This can be downloaded from `http://www.hpl.hp.com/research/linux/libunwind/`.

  – gcc. The gnu c/c++ compiler for both the host and target machines. For the IA-64 machine, the gcc version 2.96 is to be used.

[5]Note that newer versions of Jikes RVM use classpath for Java libraries. However, until Jikes RVM for IA-64 is updated to the latest CVS, the oti libraries included here should be used.

A.1 Installation overview.

In order to install Jikes RVM one must adhere to the following steps.

1. Set up a working and a build directory.
2. Set various environment variables.
3. Edit environment scripts.
4. Choose a configuration and run the configuration script to write the appropriate directory and configuration specific files to the build directory.
5. Build an executable version of Jikes RVM.

A.2 Installation steps.

1. **Setup a working and a build directory.** If Jikes RVM is downloaded from Oal CVS, the working directory shall be $HOME/ravi-ia64/rvm. To set up a build directory create a new directory $HOME/ia64_builds
2. **Setup the environment variables.** You need to setup the following environment variables:
   - RVM_ROOT the directory that contains the rvm sources.
   - RVM_BUILD the directory where you would like the build process to generate an executable RVM configuration.
   - RVM_HOST_CONFIG the configuration file used to specify the software environment on which the system is generated; i.e., where the boot image is generated.

- RVM_TARGET_CONFIG the configuration file used to specify the software environment where the system support is generated; i.e., where the booter and C runtime will be generated.
- PATH your path should contain $RVM ROOT/rvm/bin in order to pick up various scripts and utilities.

We recommend you set up these variables in your shell configuration file. For example, for bash, you might insert the following into your .bashrc file.

```
export JIKES_HOME=$HOME/ravi-ia64
#define your working directory
export RVM_ROOT=$JIKES_HOME/rvm
#define your current build directory
export RVM_BUILD=$HOME/ia64-builds
export PATH=$RVM_ROOT/rvm/bin:$PATH
export RVM_HOST_CONFIG=$RVM_ROOT/rvm/config/i686-pc-linux-gnu
export RVM_TARGET_CONFIG=$RVM_ROOT/rvm/config/ia64-linux-gnu
```

Note: You should define each of these environment variables as an absolute path. The builder template expansion process will crash and burn if you use a .. in these paths. If the host machine is PowerPC, the corresponding export would be replaced with,

```
export RVM_HOST_CONFIG=$RVM_ROOT/rvm/config/
                                powerpc-unknown-linux-gnu
```

3. **Edit configuration scripts.** You must edit the scripts defined by environment variables $RVM_HOST_CONFIG and $RVM_TARGET_CONFIG to set up variables used by the installation process. For information about editing these scripts, see installation guide in the userguide supplied with Jikes RVM sources. There are three

extra fields defined for Jikes RVM on IA-64. You should insert them if they are already not present in the file defined by $RVM_TARGET_CONFIG

```
#location of the unwind library.
export UNWIND_PATH="/opt/libunwind-0.97"
export RVM_WITH_ALIGN_BIG_FIELDS=1
export RVM_WITH_ALIGN_BIG_REF_FIELDS=1
```

Performance of Jikes RVM on IA-64 will suffer badly if the last two fields are not present in the appropriate configuration file.

4. **Choose configuration and populate build directory.** On the host machine, you will use the jconfigure script (in $RVM ROOT/rvm/bin) to populate your build ($RVM BUILD) directory with files. You must first choose a RVM configuration. All the possible configurations are listed as files in the directory $RVM_ROOT/rvm/config/build. However since this port only includes a baseline compiler and not an optimizing one, one should not use any configurations that have the subword 'opt' in their name. Further, this branch of JikesRVM only supports GCTk collectors. One can choose configurations such as GCTkSemiSpaceBaseBase, GCTkRealofBaseBase or GCTkGenBaseBase. As a next step, use the jbuild script, located in the $RVM BUILD directory to get a boot image. This phase of the build process will complete with the words 'please run me on IA-64 Linux'. For more information on choosing a configuration see the installation guide in the supplied userguide.

```
% jconfigure GCTkSemispaceBaseBase
% cd $RVM_BUILD
% jbuild
```

5. **Build an executable version of RVM.** On the target machine set the appropriate environment variables and use the jbuild script , located in the $RVM BUILD directory, to build an executable system. This script now builds an executable C program

to start the RVM, and writes to the RVM boot image. The boot image now is the binary image of a ready-to-go instance of the RVM.

```
% export RVM_ROOT=$HOME/rvmRoot
% export RVM_BUILD=$HOME/rvmBuild
% export PATH $RVM_ROOT/rvm/bin:$PATH
% jbuild -booter
```

For more information on building the executable RVM, see installation guide in the supplied userguide.

A.3 Running Jikes RVM

Jikes RVM executes bytecodes from .class files. It does not compile Java source code. Therefore, all files required by your program must have already been compiled into byte-code files by a Java compiler. We recommend you use the IBM Jikes compiler.
For example, to run class foo with source code in file foo.java

```
% jikes foo.java
% rvm foo
```

The general syntax is
rvm [rvm options] class [args]
For more information on running Jikes RVM and the command line options available for it, see section ”Running RVM” in the supplied userguide.

References

[1] *Itanium Software Conventions and Architecture Guide*. Intel, 2001.

[2] *Intel Itanium Architecture Software Developers Manual, Volume 1: Application Architecture*. Intel, 2.1 edition, 2002.

[3] *Intel Itanium Architecture Software Developers Manual, Volume 2: System Architecture*. Intel, 2.1 edition, 2002.

[4] *Intel Itanium Architecture Software Developers Manual, Volume 3: Instruction Set Reference*. Intel, 2.1 edition, 2002.

[5] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the 2001 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, 2001.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[7] Sergiy Kyrylkov. Jikes Research Virtual Machine Design and Implementation of a 64-bit PowerPC port. Master's thesis, University of New Mexico, 2003.

[8] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[9] David Mosberger and Stéphanie Eranian. *IA-64 Linux kernel, Design and Implementation*. Prentice Hall, 2002.

[10] David Mosberger-Tang. libunwind-ia64. *linux man page*.

[11] Antero Taivalsaari. Implementing a Java virtual machine in the Java programming language. *Technical Report SMLI TR-98-64*, March 1998.